# Constructing Commercial Off-the-Shelf from Legacy Systems: A Conceptual Framework

Torbjorn Skarmstad
Md. Khaled Khan*
Md. Abdur Rashid**

Department of Informatics and Computer Science
Norwegian University of Science and Technology
N-7491 Dragvoll, Trondheim, Norway
Email: torbjorn.skramstad@ifi.ntnu.no

*Peninsula School of Computing and IT
Monash University
Frankston, Vic 3199 Australia
Email: khaled.khan@infotech.monash.edu.au

**Department of Information Engineering
Massey University, Albany campus
Private Bag 102904, North Shore Mail Centre
Auckland, New Zealand
Email: m.a.rashid@massey.ac.nz

**Abstract**

*The paper introduces our current research framework for the transition of functional computing units from legacy systems to a component-based software development environment. In this respect we discuss how various computing units from legacy information systems could be converted into independent software components. Based on this we formulate a framework within which extracted computing units could be gradually migrated as independent commercial off-the-shelf (COTS).*

**Keywords**

Software components, legacy systems, program structure, reverse engineering, wrapper, software architecture.

## INTRODUCTION

Component-based software development is considered as a new paradigm shift from object-oriented software development. Although the notion of components is not entirely new, its applicability is recently getting new momentum both in research and in practice. In component-based software engineering (CBSE), an information system is considered as a set of interacting separable stand-alone components (Brown and Wallnau 1998). Software components may be located in the Internet, and dynamically discovered and assembled by other components into target systems at run-time. Usually, these pieces of components exist in binary forms, and their design artifacts are not usually known to the software composer. Software composers have neither

control nor access to the detailed design artifacts of software components. The main stream research on the component technology is evolving in two major research directions: constructing reusable components, and their effective composition.

The focal point of our current on-going research is to investigate how COTS can be constructed from the existing computing units embedded in the legacy information systems. The existing legacy systems embody a collection of large properties of algorithms and functions which could be reusable in new systems development. Legacy application systems, particularly some of their functions, are too valuable to be discarded and too expensive to reproduce (Bennett 1995). The current advances of reverse engineering technology could be an ideal candidate in extracting these lower level design artifacts of legacy information systems. The approach is applied to recover the functions and procedures embedded in the large systems. Reverse engineering tools are capable of assisting programmers to better comprehend a legacy system, its behavior and its design structure.

The on-going work reported in this paper is basically based on our earlier work on software maintenance and reverse engineering technique. Our research on reverse engineering approach to software maintenance (Khan and Skramstad 1995) has prompted us to investigate the possibility to construct COTS from the output generated by reverse engineering tools applied to legacy information systems. In this direction, we propose a framework that may facilitate this objective. Reverse engineering tools produce module chart, intermodular data flow diagram, procedure call-graph, control structure, parameter listings, global data structures, shared data format and so on (Linos et al.1994; Berg and Broek 1995).

The rest of the paper is organised as follows. Section 2 describes the general concepts of component-based development approach and related issues. In section 3 a preliminary framework on a transition procedure from legacy systems to COTS is discussed. Section 4 highlights the major unresolved issues regarding the underlying technology of COTS and CBSE.

## COMPONENT-BASED TECHNOLOGY

The main objective of the component based approach is that end users and programmers would be able to enjoy the same high levels of plug-and-play or drag-and-drop application interoperability (Adler 1995). The approach is based on the adaptability and assembly of existing programming components. The central idea of component based approaches is to enable the construction of application systems based on third-party programming units whose design and implementations details are unknown (Kozacynski and Booch 1998).

Software components are mostly in binary executable forms. They can be shared across applications to reduce the redundancy. As organisations adopt component-based software engineering, it becomes essential to clearly define its characteristics, advantages and organisational impacts (Brown and Wallnau 1998). Centralised mainframe-based applications accessed via terminals over proprietary networks have given way to distributed, multi-tiered applications remotely accessible from a variety of client machines over intranets and the Internet. Although the concept of reusing software components has existed for some time, the practice only recently gained momentum.

To minimise the efforts of redesigning the same type of software functions

repeatedly, the information systems community is moving to component based development paradigm. Object-oriented programming had a lot of promises but it has never fulfilled all those promises (Kiely 1998; Szyperski 1998). It is generally argued that, objects developed in one programming language cannot be used in other programming languages within the same application domain. This is a serious constraint that developers encounter these days. There are some fundamental differences between object-oriented approaches and component based technology as component technology guarantees interoperability of application objects across programming languages and operating platforms. The traditional programming languages only support simple interconnection between programming modules through function or procedure calls (Shaw and Garlan 1996). These interconnection formalisms require prior agreement between the participating modules regarding the data format they are to share, ordering of their interconnecting parameters, module name matching, scope of the modules and their relative position in the module hierarchy and so on. However, some of these mechanisms are brought into the component composition frameworks to facilitate the component based development.

Designing information systems as a set of components such as “modules”, “packages”, “functions” is nothing new as the approach was practiced for at least last thirty years (Brown 1996). The new issues in this field are how to construct reusable independent components, and how these stand-alone components could be composed together to build a new system on a heterogeneous platform. The advancement of Web and Internet technology has prompted us to think in this direction because various components can be discovered on the Internet, and they can be assembled dynamically. This may ultimately shape the notion of distributed information paradigm. Several vendors have already developed object model standards that provide basic facilities for constructing and interconnecting components like JavaBeans from Sun and DCOM from Microsoft. There are three major players in the commercial component market: Microsoft DCOM, and ActiveX, Sun’s Enterprise JavaBeans, and OMG’s CORBA. Recent development of Common Object Request Broker Architecture (CORBA) implements the coordination infrastructure of components that formulates which components could be composed and how they should be composed. It further defines the interface mechanisms of components. CORBA provides an architectural infrastructure to support components wrapping, composition, and coordination. CORBA is viewed as a standard platform for component integration. However, the greatest challenge is to resolve problems such as how different classes of components running on different OS platform and supported by different architectures could communicate with each other and share data among different formats.

## FROM LEGACY SYSTEMS TO COTS

Legacy systems may be defined informally as “large software systems that we do not know how to cope with, but that are vital to our organisation”. Most of these systems were written years ago using outdated techniques, yet many of them continue to do useful work. Alternatives to discard the systems or redeveloping them are being explored (Bennett, 1995). One such approach is proposed in (Sneed 1995). Another promising approach is to “freese” and encapsulate the legacy component in a new implementation. Our proposed approach described below is a combination of these two approaches.

Automatic and semi-automatic reverse engineering tools are capable of producing program structure such as module structures, call structures, parameter listings of procedures and functions (Sneed 1995). These pieces of computing units provide certain services and functions to other parts of the program. In component based approach these units are scaled up according to their functions. A draft skeleton of our proposed framework is presented in Figure 1. A legacy system is reverse engineered, and the major computing units are generated by the tool as shown in Figure 1.

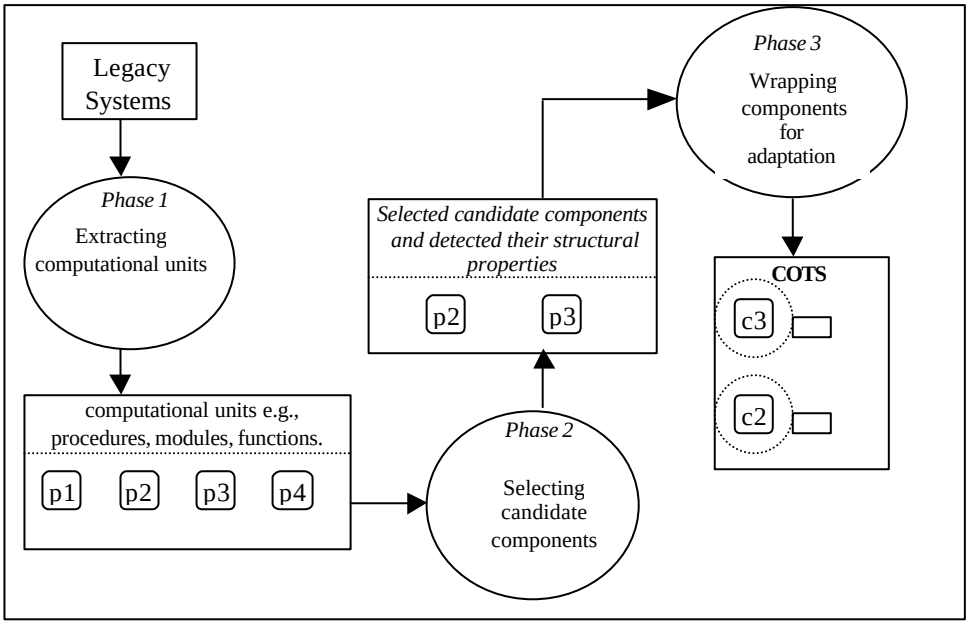


Figure 1: From Legacy systems to COTS based systems

The framework consists of three phases*: extracting computational units*, *selecting candidate components* and, *wrapping components for adaptation*. In Figure 1, the extracted computational units and components are denoted by the terms p1..p4, and c2..c3 respectively. The arrows are used to show the direction of the processing and the flow of control.

**Phase 1 : Extracting computational units**

The first phase in constructing components is to decompose a candidate legacy system into its computational units such as functions and procedures. Each of these units has structural properties in terms of parameters, data structures, control structure, and call sequences. The relationship between these structural properties in a function is identified. A function can be structurally dependent on other functions. The functions or procedures in procedure oriented language are the ideal candidates to became COTS. However, a larger chunk of computational units such as modules or sub-systems can also be converted into software components. Reverse engineering tools can be used to extract these design information. Currently, there are quite a number of reverse engineering tools available such as the CIA system (Ramamoorthy et al. 1990), CARE (Linos et al. 1994), the Act and BattleMap (Oman 1990), and VIFOR (Rajlich 1990). These tools are used to retrieve and display structural and functional dependencies of program. For example, CARE analyses C programs to extract seven different types of entities such as local and global variables, local and global constants, formal parameters, functions, and data types. The relationship between

these entities are also established by the tool such as call relationships, use of variables, and so on.

**Phase 2: Selection of candidate components**

The next phase is to identify the potential candidate components from the extracted computational units. All of the extracted units may not be qualified to become COTS. A component can be a module, a function, a procedure, or a subsystem. It is also important to classify software components into two major classes based on their functions: *domain-specific* components and *general-purpose all-domain* components. Examples of *domain specific* components are calculating overtime payment, an account transaction, and so on. On the other hand, some units are *general-purpose all-domain* units, such as finding the larger value, sorting a series of numerical values, and so on. A rigorous component selection procedure should be adapted in this respect. The selection of candidate components is based on the application domain of the computational units, and the complexity of their implementation structure and dependencies. This process includes identifying the internal data structures of the component, services it provides, its shared data formats, its dependencies on other units, and the parameters for its interfaces with associated operations. These properties and attributes of a component at this stage are usually programming language dependent. The component should be given a meaningful name which is able to reflect its services to the application designers. Figure 2 shows a schema of the non-functional attributes of a component. During the selection of component we must ensure that components are different from other kinds of reusable software units, as components are not modifiable due to their binary representation, whereas other units such as libraries, and subroutines can be modified (Krieger and Adler 1998).

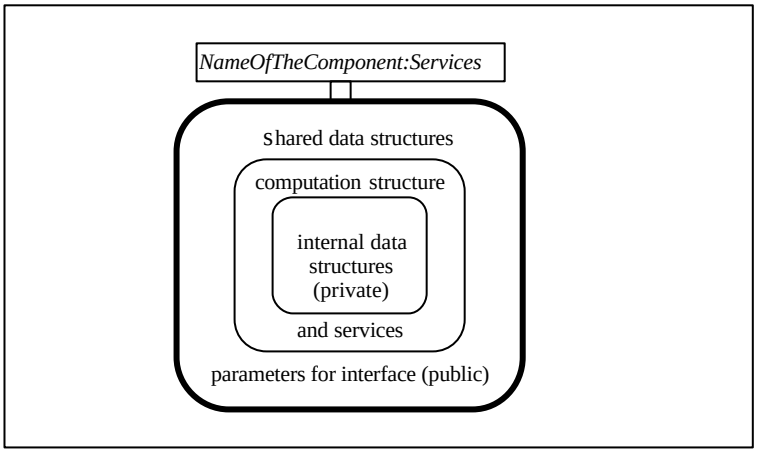


Figure 2: Language Dependent Attributes of a Component

Some computing units which have significant functionality with independent execution capability are thus preferably selected “as is” rather than "modified", and are treated as black boxes (Wallace et al. 1996). Some components have evolutionary characteristics, as they tend to update their attributes and other characteristics from time to time (Talbert 1998; Chavez et al. 1998). The issue of version control of component is beyond the scope of this paper.

**Phase 3: Wrapping components for adaptation**

In this phase, we need to provide well-defined interfaces to the selected components, and compositional formalisms for gluing units together (Shaw and Garlan 1996). The computing units of the legacy systems like modules, procedures or functions require certain changes with respect to their interfaces. However, these changes should be introduced without disrupting the services of their applications, since the semantics of their application must remain unchanged. The application domains and the functionality of the components should be protected from changes made to their interfaces (Digre 1998).

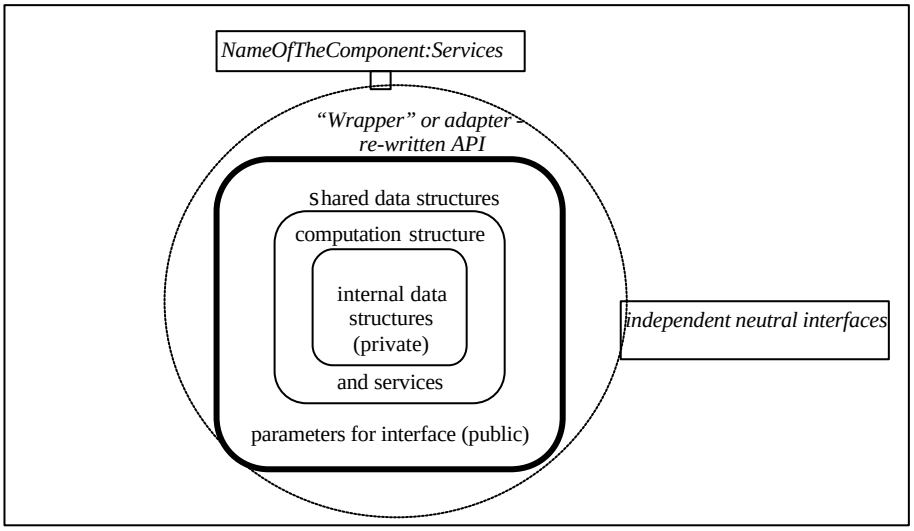


Figure 3: A Component is Encased in a "Wrapper".

Figure 3 shows a graphical representation of a "wrapped" component. The interfaces between components are expressed as application programming interfaces (API). A component can be adapted by encasing it within another component which is rewritten typically in API to provide a common interface to other components.

"Wrapping" of components is necessary to eliminate the possible architectural mismatch between components. Later the "wrapped" components are composed into a common architectural style (Brown 1996). Other components can access the services of a component through its API. API facilitate the exchanges of data between components without prior knowledge of each other's data formats. The ultimate objective is to formulate a software architecture which will be able to provide component integration in distributed and heterogeneous computing environments.

It is likely that components may have architectural mismatch as far as their interfaces are concerned. It is expected that components may be acquired from various sites with a variety of interface formats, data sharing protocols and unknown attributes with varying levels of quality. To eliminate the architectural mismatch, components must be "wrapped" in such a way that all selected components must be able to communicate with each other within a common architectural framework (Brown and Wallnau 1996). This infrastructure must provide a set of threads where components from different architecture could be plugged in to achieve a common goal. In this respect, Object Request Broker (ORB) by Object Management Group (OMG) provides the fundamental messaging mechanisms that ensure the communication

between components across heterogeneous languages, platforms and machine architectures (Thomson 1998). The interface definitions and protocols can be formulated in an Interface Definition Language (IDL). An object model architecture will ensure the application interfaces via an IDL. The interfaces of the components should be designed and implemented in such a way that they are completely independent of the component locations, their functionality and the internal implementation details of the components. Thus the behaviour of software components in a distributed setting would be ensured.

## UNRESOLVED CHALLENGES

During the construction of COTS, we need to consider to characterise the non-functional quality attributes of components such as security, reliability, scalability and so on. Particularly in distributed systems like component-based electronic commerce, it is important to consider the security properties of components. The proper specification of these quality attributes would certainly contribute in building confidence and trust on software components

Many fundamental unresolved questions should be addressed properly in this research direction as reported in (Thomson 1998). How can we derive non-functional properties from components? Is reverse engineering the only way to do this? How to maintain the composed system with changes taking place more often to components? What about the new-releases of middle-ware components more often arriving in the market? How will CBSE also benefit other areas like reusable design of all or part of a system that is represented by a set of abstract classes and the way they interact e.g., Frameworks, Patterns, OO-reuse technique? These issues should be taken under active considerations. The success of component integration depends on the performance of the hosts or servers where components reside, the visibility of the components to the application designers, the reliability and the security of the components. We believe that component based approach will soon become a promising technological advancement in effective interactions between existing independent programming units written in various languages and platforms by different developers.

## REFERENCES


Adler, R. (1995). Emerging Standards for Component Software, *IEEE Computer*, March, 68-77.

Bennett, K.(1995) Legacy Systems: Coping with Success, *IEEE Software*, Jan,19-23.

Berg, K. and Borek, P (1995) Static analysis of functional programs, *Information and Software Technology*, Elsevier Science, 37(4), 213-244.

Brown, A. (1996). Preface: Foundations for Component-Based Software Engineering, *Component-Based Software Engineering*, IEEE Computer Society Press, vii-x.

Brown, A. and Wallnau, K. (1996) Engineering of Component-Based Systems, *Component-Based Software Engineering*, IEEE Computer Society Press, 7-15.

Brown, A. and Wallnau, K. (1998) The Current State of CBSE, *IEEE Software*, September/October, 37-46.

Chavez, A., Tornabene, C. and Wiederhold (1998) Software Component Licensing: A Primer, *IEEE Software*, September/October, 47-53.

CSM *Proceedings of the International Conferences on Software Maintenance*, IEEE Computer Society press.

Digre, T. (1998) Business Object Component Architecture, *IEEE Software*, September/October, 60-69.

Khan, M. and Skramstad, T. (1995) Data Structures for Extracting and Preserving Low-Level Program Information, *Malaysian Journal of Computer Science*, Vol. 8 No.2, December, 66-79.

Kiely, D. (1998) Are Components the Future of Software?, *IEEE Computer*, February, 10-11.

Kozacynski, W. and Booch, G. (1998) Component-Based Software Engineering, *IEEE Software*, September/October, 34-36.

Krieger, D. and Adler, R. (1998) The Emergence of Distributed Component Platforms, *IEEE*

*Computer*, March, 43-53.
Linos, P., et al. (1994) Visualizing Program Dependencies: An Experimental Study, *Software-Practice and Experience*, John Wiley, Vol. 24(4), April, 387-403.
Oman, P. (1990) Maintenance Tools, *IEEE Software*, May 59-65.
Rajlich, V., Damaskinos, Linos, P., and Khorshid, W. (1990) VIFOR: A tool for Software Maintenance, *Software-Practice and Experience*, 20, 67-77.
Ramamoorthy, V., Chen, F. and Nishimoto, M (1990) The C information abstraction system, *IEEE Trans. on Software Engineering*, 16, (3) 325-334.
Shaw, A. and Garlan, D. (1996) *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, 1-242.
Sneed, H. (1995) Planning the Reengineering of Legacy Systems, *IEEE Software*, January, 24-34.
Szyperski, C. (1998) *Component Software*, Addision-Wesley.
Talbert, N. (1998) The Cost of COTS, *IEEE Computer*, June, 46-52.
Thomson, C.(1998) Workshop Reports, 1998 *Workshop on Compositional Software Architecture*, Monterey, USA. http://www.obj.com/workshops/ws9801/report.html
Wallace, E., Clements, P. C., and Wallnau, K. C. (1996) Discovering a System Modernization Decision Framework: A Case Study in Migrating to Distributed Object Technology, *Component-Based Software Engineering*, IEEE Computer Society Press, 123-123.

## COPYRIGHT